Synchronous Counting and Computational Algorithm Design

Danny Dolev
School of Engineering and Computer Science,
The Hebrew University of Jerusalem

Janne H. Korhonen
Helsinki Institute for Information Technology HIIT,
Department of Computer Science, University of Helsinki

Christoph Lenzen
Computer Science and Artificial Intelligence Laboratory, MIT

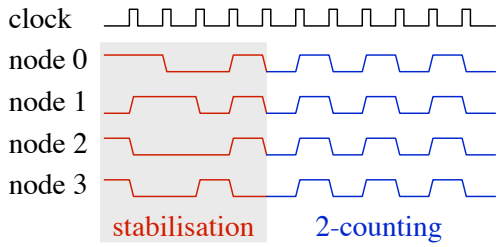
Joel Rybicki
Helsinki Institute for Information Technology HIIT,
Department of Computer Science, University of Helsinki

Jukka Suomela
Helsinki Institute for Information Technology HIIT,
Department of Computer Science, University of Helsinki

Abstract. Consider a complete communication network on n nodes, each of which is a state machine with s states. In *synchronous 2-counting*, the nodes receive a common clock pulse and they have to agree on which pulses are "odd" and which are "even". We require that the solution is *self-stabilising* (reaching the correct operation from any initial state) and it tolerates f *Byzantine failures* (nodes that send arbitrary misinformation). Prior algorithms are expensive to implement in hardware: they require a source of random bits or a large number of states s. We use computational techniques to construct very compact deterministic algorithms for the first non-trivial case of $f = 1$. While no algorithm exists for $n < 4$, we show that as few as 3 states are sufficient for all values $n \geq 4$. We prove that the problem cannot be solved with only 2 states for $n = 4$, but there is a 2-state solution for all values $n \geq 6$.

1 Introduction

Synchronous Counting. In the *synchronous C-counting* problem, n nodes have to count clock pulses modulo C. Starting from any initial configuration, the system has to *stabilise* so that all nodes agree on the clock value.

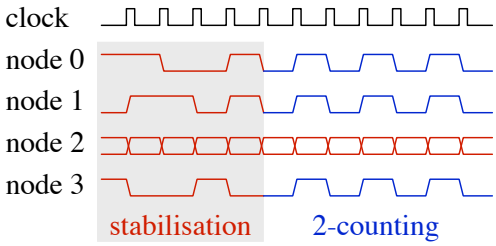


Each node is a finite state machine with s states, and after every state transition, each node *broadcasts* its current state to all other nodes—effectively, each node can see the current states of all other nodes. An algorithm specifies (1) the new state for each observed state, and (2) how to map the internal state of a node to its output.

Byzantine Fault Tolerance. In a fault-free system, the C-counting problem is trivial to solve. For example, we can designate node 0 as a leader, and then all nodes (including the leader itself) can follow the leader: if the current state of the leader is c, the new state is $c+1 \bmod C$. This algorithm will stabilise in time $t = 1$, and we only need $s = C$ different states.

However, we are interested in algorithms that tolerate *Byzantine failures*. Some number f of the nodes may be *faulty*. A faulty node may send arbitrary misinformation to non-faulty nodes, including *different* information to different nodes within the same round. For example, if we have nodes $0, 1, 2, 3$ and node 2 is faulty, node 0 might observe the state vector $(0, 1, 1, 1)$, while node 1 might observe the state vector $(0, 1, 0, 1)$.

Our goal is to design an algorithm with the following guarantee: even if we have up to f faulty nodes, no matter what the faulty nodes do, the system will stabilise so that after t rounds all non-faulty nodes start to count clock pulses consistently modulo C. We will give a formal problem definition in Section 4.



Contributions. Both randomised and deterministic algorithms for synchronous counting have been presented in the literature (see Section 2). However, prior algorithms tend to be expensive to implement in hardware: they require a source of random bits or complicated circuitry.

In this work, we use a single parameter s, the number of states per node, to capture the complexity of an algorithm. We employ *computational* techniques to design 2-counting algorithms that have the smallest possible number of states. Our focus is on the first non-trivial case of $f = 1$.

The case of $n = 1$ is trivial, and by prior work it is known that there is no algorithm for $1 < n < 4$. We give a detailed analysis of 2-counting for $n \geq 4$:

- there is no deterministic algorithm for $f = 1$ and $n = 4$ with $s = 2$ states,
- there is a deterministic algorithm for $f = 1$ and $n \geq 4$ with $s = 3$ states,
- there is a deterministic algorithm for $f = 1$ and $n \geq 6$ with $s = 2$ states.

With such a small state space, the algorithms are easy to implement in hardware. For example, a straightforward implementation of our algorithm for $f = 1$, $n = 4$, and $s = 3$ requires just 2 bits of storage per node, and a simple lookup table with 81 entries.

Our results are related to synchronous 2-counting, but we can compose b copies of a 2-counter to construct a 2^b-counter (see Section 3 for details).

Structure. Section 2 covers related work and Section 3 discusses applications of synchronous 2-counters. Section 4 gives a formal definition of the problem, and Section 5 gives a graph-theoretic interpretation that is helpful in the analysis of counting algorithms. In Section 6 we show that we can increase n for free, without affecting the parameters f, s, or t; this enables us to focus on small values of n. Section 7 demonstrates the use of computers in algorithm design, and Section 8 shows how we

can use computers to construct compact proofs of the *non-existence* of algorithms for given values of n, f, and s.

2 Related Work

Randomised Algorithms. Randomised algorithms for synchronous 2-counting are known, with different time–space tradeoffs.

The algorithm by Dolev and Welch [8] requires only $s = 3$ states, but the stabilisation time is $t = 2^{O(f)}$. Here we are assuming that $n = O(f)$; for a large n, we can run the algorithm with $O(f)$ nodes only and let the remaining nodes follow the majority.

The algorithm by Ben-Or et al. [2] stabilises in expected constant time. However, it requires $\Omega(2^f)$ states and private channels (i.e., the adversary has limited information on the system's state).

Deterministic Algorithms. The fastest known deterministic algorithm is due to Dolev and Hoch [6], with a stabilisation time of $O(f)$. However, the algorithm is not well suited for a hardware implementation. It uses as a building block several instances of algorithms that solve the Byzantine consensus problem—a non-trivial task in itself. The number of states is also large, as some storage is needed for each Byzantine consensus instance.

Consensus Lower Bounds. *Binary consensus* is a classical problem that has been studied in the context of Byzantine fault tolerance; see, e.g., the textbook by Lynch [11] for more information. It is easy to show that synchronous 2-counting is at least as difficult to solve as binary consensus.

Lemma 1. *If we have a 2-counting algorithm $\mathcal{A}$ that stabilises in time t, we can design an algorithm that solves binary consensus in time t, for the same parameters n and f.*

Proof. We can find configurations $\mathbf{x}(0)$ and $\mathbf{x}(1)$ with the following properties:

- For any $a = 0, 1$ and $j = 0, 1, 2, \ldots$, if we initialise the system with configuration $\mathbf{x}(a)$ and run $\mathcal{A}$ for j rounds, all nodes output $(a + j) \bmod 2$.

In essence, $\mathbf{x}(0)$ and $\mathbf{x}(1)$ are some examples of configurations that may occur during 2-counting.

First assume that t is even. Each node i receives its input a for the binary consensus problem. We use the element i of $\mathbf{x}(a)$ to initialise the state of node i. Then we run $\mathcal{A}$ for t rounds. Finally, the output of algorithm $\mathcal{A}$ forms the output of the binary consensus instance. To see that the algorithm is correct, we make the following observations:

1. All non-faulty nodes produce the same output at time t, regardless of the input.
2. If all inputs had the same value a, we used $\mathbf{x}(a)$ to initialise all nodes, and hence the final output is a.

For an odd t, we can use the same approach if we complement the inputs. In summary, $\mathcal{A}$ can be used to solve binary consensus in time t. □

Now we can invoke the familiar lower bounds related to the consensus problem:

- no algorithm can tolerate $f \geq n/3$ failures [13],
- no deterministic algorithm can solve the problem in $t < f + 1$ rounds [9].

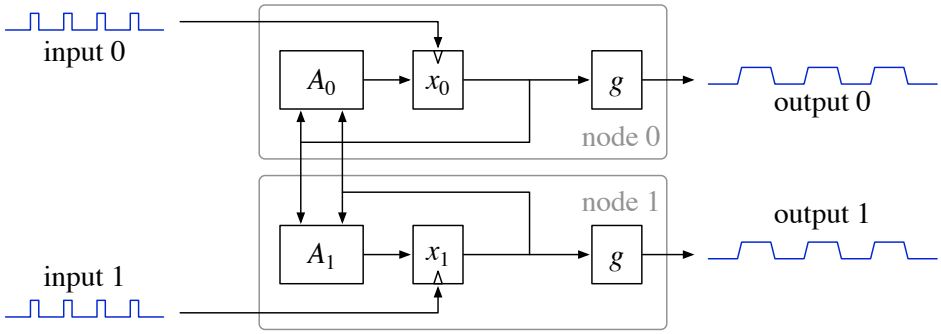


Figure 1: A 2-counter for $n = 2$, viewed as an electronic circuit.

Pulse Synchronisation. Both 2-counting and *pulse synchronisation* [5, 8] have a superficially similar goal: produce well-separated, (approximately) synchronised clock pulses in a distributed system in a fault-tolerant manner. However, there are also many differences: in pulse synchronisation the task is to construct a clock pulse without any external reference, while in 2-counting we are given a reference clock and we only need to construct a clock that ticks at a slower rate. Also the models of computation differ—for pulse synchronisation, a relevant model is an asynchronous network with some bounds on propagation delays and clock drifts.

A 2-counting algorithm does not solve the pulse synchronisation problem, and a pulse synchronisation algorithm does not solve the 2-counting problem. However, if one is designing a distributed system that needs to produce synchronised clock ticks in a fault-tolerant manner, either of the approaches may be applicable.

3 Applications

Counters as Frequency Dividers. We can visualise a C-counter as an electronic circuit that consists of n components (nodes); see Figure 1. Each node i has a register x_i that stores its current state—one of the values $0, 1, \ldots, s-1$. There is a logical circuit g that maps the current state to the output, and another logical circuit A_i that maps the current states of all nodes to the new state of node i. At each rising edge of the clock pulse, register x_i is updated.

If the clock pulses are synchronised, regardless of the initial states of the registers, after t clock pulses the system has stabilised so that the outputs are synchronised and they are incremented (modulo C) at each clock pulse.

In particular, if we have an algorithm for 2-counting, it can be used as a *frequency divider*: given synchronous clock pulses at rate 1, it produces synchronous clock pulses at rate $1/2$.

From 2-Counters to C-counters. We can compose b layers of 2-counters to build a clock that counts modulo 2^b; see Figure 2. A composition of self-stabilising algorithms is self-stabilising [7]. For the purposes of the analysis, we can wait until layer $i-1$ stabilises, use this as the initial state of layer i, and then argue that the nodes on layer i receive a synchronous clock pulse and hence they will eventually stabilise.

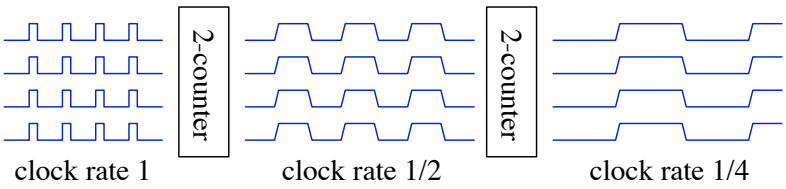


Figure 2: Composition of 2-counters.

Fault-Tolerant Counting. Assume that we have n nodes that are supposed to count events that can be observed by all nodes. However, some nodes might occasionally miss some events, and we may have transient failures. A synchronous C-counter provides an elegant solution to the problem: observations of the events are interpreted as clock pulses. A node that misses an event is merely one kind of faulty behaviour; if there are at most f such nodes, all other nodes will keep counting correctly. Moreover, if we have at most f nodes whose current counter values are incorrect, these nodes will eventually catch up with the others.

Counters in Mutual Exclusion. With a C-counter we can implement *mutual exclusion* and *time division multiple access* in a fairly straightforward manner. If we have $C = n$ nodes and one shared resource (e.g., a transmission medium), we can let node i to access the resource when its own counter has value i. Care is needed with the actions of faulty nodes, though—for further information on achieving *fault-tolerant* mutual exclusion, see, e.g., Moscibroda and Oshman [12]. Again 2-counting is of particular interest, as it may be leveraged by more complex mutual exclusion algorithms.

4 Problem Formulation

We will now formalise the C-counting problem and introduce the definitions that we will use in this work. Throughout this work, we will follow the convention that nodes, states, and time steps are indexed from 0. We use the notation $[k] = \{0, 1, \ldots, k-1\}$.

Simplifications. As our focus is on 2-counters, we will now fix $C = 2$; the definitions are straightforward to generalise.

In prior work, algorithms have made use of a function that maps the internal state x_i of a node to its output $g(x_i)$. However, in this work we do not need any such mapping: for our positive results, an identity mapping is sufficient, and for the negative result, we study the case of $s = 2$ which never benefits from a mapping. Hence we will now give a formalisation that omits the output mapping.

Algorithms. Fix the following parameters:

- n = the number of nodes,
- f = the maximum number of faulty nodes,
- s = the number of internal states.

An algorithm $\mathbf{A}$ specifies a *state transition* function $A_i \colon [s]^n \to [s]$ for each node $i \in [n]$. Here $[s]^n$ is the set of *observed configurations* of the system.

Projections. Let $F \subseteq [n]$, $|F| \leq f$ be the set of *faulty* nodes. We define the projection π_F as follows: for any observed configuration $\mathbf{s}$, let $\pi_F(\mathbf{s})$ be a vector $\mathbf{x}$ such that $x_i = *$ if $i \in F$ and $x_i = s_i$ otherwise. For example,

$$\pi_{\{2,4\}}((0,1,0,1,1)) = (0,1,*,1,*).$$

This gives us the set $V_F = \pi_F([s]^n)$ of *actual configurations*. Two actual configurations are particularly important:

$$\mathbf{0}_F = \pi_F((0,0,\dots,0)), \quad \mathbf{1}_F = \pi_F((1,1,\dots,1)).$$

Executions. Let $\mathbf{x}, \mathbf{y} \in V_F$. We say that configuration $\mathbf{y}$ is *reachable* from $\mathbf{x}$ if for each non-faulty node $i \notin F$ there exists some observed configuration $\mathbf{u}_i \in [s]^n$ satisfying $\pi_F(\mathbf{u}_i) = \mathbf{x}$ and $A_i(\mathbf{u}_i) = y_i$. Intuitively, the faulty nodes can feed such misinformation to node i that it chooses to switch to state y_i. We emphasise that $\mathbf{u}_i$ may be different for each i; the misinformation need not be consistent.

An *execution* of an algorithm $\mathbf{A}$ for given set of faulty nodes F is an infinite sequence of actual configurations $X = (\mathbf{x}^0, \mathbf{x}^1, \mathbf{x}^2, \dots)$ such that $\mathbf{x}^{r+1}$ is reachable from $\mathbf{x}^r$ for all r.

Stabilisation. For an execution $X = (\mathbf{x}^0, \mathbf{x}^1, \mathbf{x}^2, \dots)$, define its t-tail

$$X[t] = (\mathbf{x}^t, \mathbf{x}^{t+1}, \mathbf{x}^{t+2}, \dots).$$

We say that X *stabilises in time* t if one of the following holds:

$$X[t] = (\mathbf{0}_F, \mathbf{1}_F, \mathbf{0}_F, \dots) \quad \text{or} \quad X[t] = (\mathbf{1}_F, \mathbf{0}_F, \mathbf{1}_F, \dots).$$

We say that an algorithm $\mathbf{A}$ *stabilises in time* t if for any set of faulty nodes F with $|F| \leq f$, all executions of $\mathbf{A}$ stabilise in time t.

5 Projection Graphs

Before discussing how to *find* an algorithm (or prove that an algorithm does not exist), let us first explain how we can *verify* that a given algorithm is correct. Here the concept of a *projection graph* is helpful.

Fix the parameters s, n, and f, and consider a candidate algorithm $\mathbf{A}$ that is supposed to solve the 2-counting problem. For each set $F \subseteq [n]$ of faulty nodes, construct the directed graph $G_F(\mathbf{A}) = (V_F, R_F(\mathbf{A}))$ as follows:

1. The set of nodes V_F is the set of actual configurations.
2. There is an edge $(\mathbf{u}, \mathbf{v}) \in R_F(\mathbf{A})$ if configuration $\mathbf{v} \in V_F$ is reachable from configuration $\mathbf{u} \in V_F$. In general, this may produce self-loops.

Note that the outdegree of each node in $G_F(\mathbf{A})$ is at least 1. Directed walks in $G_F(\mathbf{A})$ correspond to possible executions of algorithm $\mathbf{A}$, for this set F of faulty nodes. To verify the correctness of algorithm $\mathbf{A}$, it is sufficient to analyse the projection graphs G_F. The following lemmas are straightforward consequences of the definitions.

Lemma 2. *Algorithm* $\mathbf{A}$ *stabilises in some time* t *iff for every* F*, graph* $G_F(\mathbf{A})$ *contains exactly one directed cycle,* $\mathbf{0}_F \mapsto \mathbf{1}_F \mapsto \mathbf{0}_F$*.*

Lemma 3. *Algorithm* $\mathbf{A}$ *stabilises in time* t *iff the following holds for all* F*:*

1. *In* $G_F(\mathbf{A})$*, the only successor of* $\mathbf{0}_F$ *is* $\mathbf{1}_F$ *and vice versa.*
2. *In* $G_F(\mathbf{A})$*, every directed walk of length* t *reaches node* $\mathbf{0}_F$ *or* $\mathbf{1}_F$*.*

Lemma 4. *Let* $\mathbf{A}$ *be an algorithm. Consider any four configurations* $\mathbf{x}, \mathbf{u}, \mathbf{v}, \mathbf{w} \in V_F$ *with the following properties:* $(\mathbf{x}, \mathbf{u}) \in R_F(\mathbf{A})$*,* $(\mathbf{x}, \mathbf{v}) \in R_F(\mathbf{A})$*, and* $w_i \in \{u_i, v_i\}$ *for each* $i \notin F$*. Then* $(\mathbf{x}, \mathbf{w}) \in R_F(\mathbf{A})$*.*

6 Increasing the Number of Nodes

It is not obvious how to use computational techniques to design an algorithm that solves the 2-counting problem for a fixed $f = 1$ but arbitrary $n \geq 4$. However, as we will show next, we can generalise any algorithm so that it solves the same problem for a larger number of nodes, without any penalty in time or space complexity. Therefore it is sufficient to design an algorithm for the special case of $f = 1$ and $n = 4$.

Lemma 5. *Fix* $n \geq 4$*,* $f < n/2$*,* $s \geq 2$*, and* $t \geq 1$*. Assume that* $\mathbf{A}$ *is an algorithm that solves the* 2*-counting problem for* n *nodes, out of which at most* f *are faulty, with stabilisation time* t *and with* s *states per node. Then we can design an algorithm* $\mathbf{B}$ *that solves the* 2*-counting problem for* $n + 1$ *nodes, out of which at most* f *are faulty, with stabilisation time* t *and with* s *states per node.*

Proof. The claim would be straightforward if we permitted the stabilisation time of $t + 1$. However, some care is needed to avoid the loss of one round.

We take the following approach. Let p be a projection that removes the last element from a vector, for example, $p((a, b, c)) = (a, b)$. In algorithm $\mathbf{B}$, nodes $i \in [n]$ simply follow algorithm $\mathbf{A}$, ignoring node n:

$$B_i(\mathbf{u}_i) = A_i(p(\mathbf{u}_i)).$$

Node n tries to predict the majority of nodes $0, 1, \ldots, n-1$, i.e., what most of them are going to output after this round:

- Assume that node n observes a configuration $\mathbf{u}_n$. For each $i \in [n]$, define $h_i = A_i(p(\mathbf{u}_n))$. If a majority of the values h_i is 1, then the new state of node n is also 1; otherwise it is 0.

To prove that the algorithm is correct, fix a set $F \subseteq [n+1]$ of faulty nodes, with $|F| \leq f$. Clearly, all nodes in $[n] \setminus F$ will start counting correctly at the latest in round t. Hence any execution of $\mathbf{B}$ with $n \in F$ trivially stabilises within t rounds; so we focus on the case of $F \subseteq [n]$, and merely need to show that also node n counts correctly.

Fix an execution $X = (\mathbf{x}^0, \mathbf{x}^1, \ldots)$ of $\mathbf{A}$, and a point of time $r \geq t$. Consider the state vector $\mathbf{x}^{r-1}$. By assumption, $\mathbf{A}$ stabilises in time t. Hence the successors of $\mathbf{x}^{r-1}$ in the projection graph must be in $\{\mathbf{0}_F, \mathbf{1}_F\}$.

The key observation is that only one of the configurations $\mathbf{0}_F$ and $\mathbf{1}_F$ can be the successor of $\mathbf{x}^{r-1}$. Otherwise Lemma 4 would allow us to construct another state that is a successor of $\mathbf{x}^{r-1}$, contradicting the assumption that $\mathbf{A}$ stabilises.

We conclude that for all rounds $r \geq t$ and all nodes $i \in [n] \setminus F$, the value h_i is independent of the states communicated by nodes in F. Since the values h_i are identical and $n - f > f$, node n attains the same state as other correct nodes in rounds $r \geq t$. $\square$

7 Computer-Designed Algorithms

In principle, we could now attempt to use a computer to tackle our original problem. By the discussion of Section 6, it suffices to discover an algorithm with the smallest possible s for the special case of $n = 4$ and $f = 1$. We could try increasing values of $s = 2, 3, \ldots$. Once we have fixed n, f, and s, the problem becomes finite: an algorithm is a lookup table with $\ell = ns^n$ entries, and hence there are s^ℓ candidate algorithms to explore. For each candidate algorithm, we could use the projection graph approach of Section 5 to quickly reject any invalid algorithm.

Unfortunately, the search space is huge. As we will see, there is no algorithm with $n = 4$ and $s = 2$. For $n = 4$ and $s = 3$, we have approximately 10^{154} candidates. We use two complementary approaches to tackle the task:

1. Narrow down the search space by considering restricted classes of algorithms.
2. Encode the problem as a Boolean formula and apply SAT solvers.

While SAT solvers are not a panacea, it is not uncommon to see modern general-purpose SAT solvers outdo carefully engineered application-specific algorithms.

Cyclic Algorithms. We will consider two classes of algorithms—general algorithms (without any restrictions) and *cyclic* algorithms. We say that algorithm $\mathbf{A}$ is cyclic if

$$A_i((x_i, x_{i+1}, \ldots x_{n-1}, x_0, x_1, \ldots, x_{i-1})) = A_0((x_0, x_1, \ldots, x_{n-1}))$$

for all i and all $\mathbf{x}$. That is, a cyclic algorithm is invariant under cyclic renaming of the nodes.

There is no a priori reason to expect that the most efficient algorithms are cyclic. However, cyclic algorithms have many attractive features: for example, in a hardware implementation of a cyclic algorithm we only need to take n copies of identical modules. Furthermore, the search space is considerably smaller: we only need to define transition function A_0. For $n = 4$ and $s = 3$, we have approximately 10^{38} candidate algorithms.

Cyclic algorithms are also much easier to verify. The projection graphs $G_F(\mathbf{A})$ are isomorphic for all $|F| = 1$ and hence it is sufficient to check one of them.

Encoding. At a high level, we take the following approach.

1. Fix the parameters s, n, f, t, and the algorithm family (cyclic or general).
2. Construct a Boolean circuit $\mathcal{C}$ that verifies whether a given algorithm $\mathbf{A}$ is correct. The circuit receives the transition table of $\mathbf{A}$ encoded as a binary string.
3. Translate circuit $\mathcal{C}$ to an equivalent Boolean formula $\mathcal{F}$.
4. Use SAT solvers to find a satisfying assignment $\mathbf{a}$ of $\mathcal{F}$.
5. Translate $\mathbf{a}$ to an algorithm $\mathbf{A}$ that passes the verification of circuit $\mathcal{C}$.

In essence, circuit $\mathcal{C}$ applies Lemma 3 to verify $\mathbf{A}$. More concretely, it is a hard-wired implementation of a computer program that performs the following steps (in parallel for all possible sets F):

1. Construct the projection graph $G_F(\mathbf{A})$.
2. Verify that there are no self-loops in G_F.
3. Verify that the only successor of $\mathbf{0}_F$ is $\mathbf{1}_F$ and vice versa.

Parameters						SAT instance		Solver running time (s)		
s	n	f	t	family	b	variables	clauses	picosat	lingeling	plingeling
2	6	1	8		384	78546	336098	—	—	140000
2	7	1	8	cyclic	128	37230	171626	—	47	48
2	8	1	4	cyclic	256	79423	436929	410	17	12
3	4	1	7	cyclic	129	10338	42030	4	3	3
3	5	1	4		1926	374871	1712779	—	—	140000
3	5	1	6	cyclic	386	66793	304091	—	3000	1200
3	6	1	3	cyclic	1156	319726	1753824	11000	450	530
4	4	1	5	cyclic	512	62272	269892	64000	320	77
4	5	1	5	cyclic	2048	760892	3691498	—	41000	25000

Table 1: Positive results. The size of the search space is approximately 2^b.

4. For each $d = 1, 2, \ldots, t$, find the subset $B_F(d) \subseteq V_F$ of states with the following property: for each $\mathbf{u} \in B_F(d)$ there is a directed walk of length d in G_F that starts from $\mathbf{u}$ and does not traverse $\mathbf{0}_F$ or $\mathbf{1}_F$.
5. Verify that set $B_F(t)$ is empty.

For cyclic algorithms, we identify equivalent transitions in the input and simplify the circuit accordingly.

Results. The Boolean circuits were constructed with a program written in C++. We used Junttila's **bc2cnf** tool, version 0.35 [10] to convert the circuit to a SAT instance, and then we experimented with two SAT solvers: **lingeling**, version ala-b02aa1a [4], and **picosat**, version 954 [3]. We ran our experiments on a computing cluster in which each node had 2 processors (Intel Xeon E5540), 2×4 cores, and 32 GB RAM. In this environment **plingeling**, the parallel version of **lingeling**, automatically uses up to 8 threads and up to 16 GB memory.

The positive results are reported in Table 1, along with some statistics. The b column indicates how many bits are needed to encode an algorithm—equivalently, it is the base-2 logarithm of the size of the search space, rounded up.

The key findings are a cyclic algorithm for $s = 3$, $n = 4$, and $f = 1$, and a non-cyclic algorithm for $s = 2$, $n = 6$, and $f = 1$. The table also gives example of space-time tradeoffs: we can often obtain faster stabilisation if we use a larger number of states.

For the sake of comparison, we note that the *fastest* deterministic algorithm from prior work [6] stabilises in time $t = 13$ for $f = 1$ and it requires a large state space. Our algorithms achieve the stabilisation time of $t = 5$ for $s = 4$ and $t = 7$ for $s = 3$.

Machine-readable versions of all positive results, together with a Python script that can be used to verify the correctness of the algorithms, are freely available online [1].

8 Computer-Designed Lower Bounds

We can use the approach of Section 7 to discover not only positive but also negative results. If we use a SAT encoding that applies Lemma 2 instead of Lemma 3, we can use the same approach to prove, for example, that the case of $s = 2$, $n < 6$, and $f = 1$ is not solvable for any stabilisation time t.

However, such an approach would not admit easy independent verification. It is true that modern SAT solvers are able to produce, e.g., resolution proofs showing the unsatisfiability of the propositional formula. However, these proofs tend to be long, and more importantly, require us to prove that (1) the encoding exactly captures the problem at hand and (2) the computer program that outputs the formula is correct. Hence we apply different techniques.

Using C++, we implemented a backtracking algorithm that analyses the case of $s = 2$, $n = 4$, and $f = 1$. The algorithm produces a proof that shows that there is no solution.

The proof is a case analysis with 106 cases, each of them easy to verify. A machine-readable version of the proof is freely available online [1]. We have also provided a simple Python script that can be used to check that the proof is indeed correct: all possible cases are covered and the contradiction that we exhibit for each case is verified.

While it is straightforward to construct *some* proof (in principle, an exhaustive enumeration of all candidate algorithms would suffice), it is difficult to come up with a *small* proof that is short enough for a (very patient) human being to verify. Briefly, the key milestones are as follows.

1. An exhaustive enumeration: 2^{64} cases.
2. Trivial cases eliminated: 2^{32} cases.
3. A straightforward backtracking search: ≈ 10000 cases.
4. A heuristic rule that attempts to find the best possible branching order for the backtracking search: 243 cases.
5. Merging cases that can be covered with a single proof: 106 cases.

While a long proof is easy to construct, the short proofs required extensive computations in a cluster environment (in total several years of CPU time).

9 Conclusions

In this work, we have used computational techniques to study the synchronous counting problem. At first sight the problem is not well-suited for computational algorithm design—we need to reason about stabilisation from any given starting configuration, for any adversarial behaviour, in a system with arbitrarily many nodes. Nevertheless, we have demonstrated that computational techniques can be used in this context, both to discover novel algorithms and to prove lower-bound results.

Our computational results were constructed with a fairly complicated toolchain. However, the end results are compact, machine readable, and easy to verify with a straightforward script.

Our algorithms outperform the best human-designed algorithms: they are deterministic, small ($2 \leq s \leq 3$), fast ($3 \leq t \leq 8$), and easy to implement in hardware or in software—a small lookup table suffices. In summary, our work leaves very little room for improvement in the case of $f = 1$. The general case of $f > 1$ is left for future work; we are optimistic that the algorithms designed in this work can be used as subroutines to construct algorithms that tolerate a larger number of failures.

Acknowledgments. Many thanks to Matti Järvisalo for discussions and advice related to SAT encoding and SAT solvers.

Danny Dolev is Incumbent of the Berthold Badler Chair in Computer Science. This work was supported in part by the Israeli Centers of Research Excellence (I-CORE) program (Center No. 4/11), the Israeli Ministry of Science and Technology (grant 3/9778), the Academy of Finland (grants 132380 and 252018), the Research Funds of the University of Helsinki, the Helsinki Doctoral Programme in Computer Science – Advanced Computing and Intelligent Systems, the Swiss Society of Friends of the Weizmann Institute of Science, and the German Research Foundation (DFG).

References

[1] Computer-generated proofs. https://github.com/suomela/counting (primary), https://bitbucket.org/suomela/counting (backup).

[2] Michael Ben-Or, Danny Dolev, and Ezra N. Hoch. Fast self-stabilizing Byzantine tolerant digital clock synchronization. In *Proc. 27th Symposium on Principles of Distributed Computing (PODC 2008)*, pages 385–394, New York, 2008. ACM Press. doi:10.1145/1400751.1400802.

[3] Armin Biere. PicoSAT essentials. *Journal on Satisfiability, Boolean Modeling and Computation*, 4:75–97, 2008.

[4] Armin Biere. Lingeling and friends entering the SAT challenge 2012. In *Proceedings of SAT Challenge 2012; Solver and Benchmark Descriptions*, volume B-2012-2 of *Department of Computer Science Series of Publications B*, pages 33–34. University of Helsinki, 2012.

[5] Ariel Daliot, Danny Dolev, and Hanna Parnas. Self-stabilizing pulse synchronization inspired by biological pacemaker networks. In *Proc. 6th Symposium on Stabilization, Safety, and Security of Distributed Systems (SSS 2003)*, volume 2704 of *LNCS*, pages 32–48, Berlin, 2003. Springer. doi:10.1007/3-540-45032-7_3.

[6] Danny Dolev and Ezra N. Hoch. On self-stabilizing synchronous actions despite Byzantine attacks. *Distributed Computing*, 4731:193–207, 2007. doi:10.1007/978-3-540-75142-7_17.

[7] Shlomi Dolev. *Self-Stabilization*. The MIT Press, Cambridge, MA, 2000.

[8] Shlomi Dolev and Jennifer L. Welch. Self-stabilizing clock synchronization in the presence of Byzantine faults. *Journal of the ACM*, 51(5):780–799, 2004. doi:10.1145/1017460.1017463.

[9] Michael J. Fischer and Nancy A. Lynch. A lower bound for the time to assure interactive consistency. *Information Processing Letters*, 14(4):183–186, 1982. doi:10.1016/0020-0190(82)90033-3.

[10] Tommi Junttila and Ilkka Niemelä. Towards an efficient tableau method for Boolean circuit satisfiability checking. In *Proc. 1st Conference on Computational Logic (CL 2000)*, volume 1861 of *LNCS*, pages 553–567, Berlin, 2000. Springer. doi:10.1007/3-540-44957-4_37.

[11] Nancy A. Lynch. *Distributed Algorithms*. Morgan Kaufmann, San Francisco, 1996.

[12] Thomas Moscibroda and Rotem Oshman. Resilience of mutual exclusion algorithms to transient memory faults. In *Proc. 30th Symposium on Principles of Distributed Computing (PODC 2011)*, pages 69–78, New York, 2011. ACM Press. doi:10.1145/1993806.1993817.

[13] Marshall C. Pease, Robert E. Shostak, and Leslie Lamport. Reaching agreement in the presence of faults. *Journal of the ACM*, 27(2):228–234, 1980. doi:10.1145/322186.322188.

A Algorithm Listings

In this appendix, we give two examples of our algorithms—machine-readable versions of all algorithms, verification code, and some illustrations are available online [1].

Table 2 gives a cyclic algorithm for $n = 4$. The rows are labelled with (x_0, x_1), the columns are labelled with (x_2, x_3), and the values indicate $A_0((x_0, x_1, x_2, x_3))$, that is, the new state of the first node in the observed configuration $\mathbf{x}$.

Table 3 shows a non-cyclic algorithm for $n = 6$. Again, the rows are labelled with the first half (x_0, x_1, x_2) of the observed state $\mathbf{x}$ and the columns are labelled with the second half (x_3, x_4, x_5) of the observed state $\mathbf{x}$. The values show the new state for each node: $A_0(\mathbf{x}), A_1(\mathbf{x}), \ldots, A_5(\mathbf{x})$.

	00	01	02	10	11	12	20	21	22
00	1	1	1	1	2	1	1	1	1
01	1	1	1	0	2	0	1	1	0
02	1	1	1	1	0	0	1	1	1
10	1	0	0	1	0	0	1	0	0
11	0	0	0	0	0	0	0	0	0
12	0	0	0	0	0	0	0	0	0
20	1	1	1	1	1	1	1	1	1
21	1	1	1	1	0	0	1	1	0
22	1	1	1	0	0	0	1	1	1

Table 2: Cyclic algorithm for $s = 3$, $n = 4$, $f = 1$, and $t = 7$.

	000	001	010	011	100	101	110	111
000	111111	111111	111111	110101	111011	111111	111111	110101
001	111111	110111	111101	000001	111011	100011	101001	000100
010	111111	111111	111111	110101	001000	001000	001000	000000
011	101111	100011	101001	000000	001000	000000	001000	000000
100	111111	011100	110101	000000	111111	011100	100011	000000
101	110111	000000	000000	000000	100011	000000	000000	000000
110	111111	011100	100011	000000	001000	001000	000000	000000
111	100111	000000	000000	000000	000000	000000	000000	000000

Table 3: Algorithm for $s = 2$, $n = 6$, $f = 1$, and $t = 8$.